Multi-Core Memory Hierarchies

Lecture 10: Memory Access Scheduling

Tong Dong

http://mprc.pku.edu.cn/~tongdong/MMH
In the last Lecture

- The L1 Cache of SM in GPGPU
  - Large cache line
  - Small cache capacity
  - Shared by thousands of threads

- GPU Cache Management
  - Replacement: PDP or frequency-based
    - Intra-warp locality
    - Inter-warp locality
  - Bypassing: PDP and HW/SW
  - Prefetching
    - simple stride prefetchers with throttling and scheduling
    - Pre-execution

- GPU Warp Scheduling
  - Scheduling: Two-level, locality-aware, prefetching-aware, critical-aware
  - Throttling: locality/thrashing HW monitors and SW profiling
Multi-Core Cache Hierarchies

- Miss Ratio: \( m_{Li} = \left( \frac{C_{Li}}{I(NT)\beta_{Li}} \right)^{1-\alpha_{Li}} \)

- Average Access Time:
  \[ T_{Li} = (1 - m_{Li})T^{hit}_{Li} + m_{Li}T_{Li-1} \]

<table>
<thead>
<tr>
<th>Topics</th>
<th>Locality</th>
<th>Interference</th>
<th>Capacity</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement/Bypassing</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUCA</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Associativity</td>
<td></td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetching</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Compression</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>3D DRAM</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Scheduling/Throttling</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>
“It’s interference, stupid!”

Locality: Temporal locality, Spatial locality, Value locality
Parallelism: Memory Interleaving

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Capacity</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>16TB</td>
<td>~10ms</td>
</tr>
<tr>
<td>NAND</td>
<td>128GB</td>
<td>~50us</td>
</tr>
<tr>
<td>NVM</td>
<td>4~64GB</td>
<td>~70ns</td>
</tr>
<tr>
<td>SRAM/eDRAM</td>
<td>8MB</td>
<td>~15ns</td>
</tr>
<tr>
<td>SRAM</td>
<td>32/256KB</td>
<td>~2/7ns</td>
</tr>
<tr>
<td>Register</td>
<td>1KB</td>
<td>~0.3ns</td>
</tr>
</tbody>
</table>
DRAM Basic

- **Average Access Time**
  \[ T_{LLC} = (1 - m_{LLC})T_{LLC}^{hit} + m_{LLC}T_{off-chip} \]

- **Memory Controller Delay**

- **DRAM latency**
  - \( t_{RP} + t_{RCD} + t_{CL} (PRE + RAS + CAS) \)
  - Row-buffer hit: only \( t_{CL} (t_{CAS}) \)

DRAM Basic

- **Average Access Time**
  \[ T_{LLC} = (1 - m_{LLC})T_{hit} + m_{LLC}T_{off-chip} \]

- **Memory Controller Delay**

- **DRAM latency**
  - \( t_{RP} + t_{RCD} + t_{CL} \) (PRE + RAS + CAS)
  - Row-buffer hit: only \( t_{CL} \) (tCAS)

---

DRAM Basic

[Diagram showing the basic components of DRAM (Dynamic Random Access Memory), including Row Decoder, Column Decoder, Sense Amps, Data In/Out Buffers, and Memory Array.]
Basics

[PRECHARGE and] ROW ACCESS

AKA: OPEN a DRAM Page/Row
or
ACT (Activate a DRAM Page/Row)
or
RAS (Row Address Strobe)
Basics

COLUMN ACCESS

CPU

MEMORY CONTROLLER

Column Decoder

Data In/Out Buffers

Sense Amps

... Bit Lines...

Row Decoder

... Word Lines...

Memory Array

READ Command

or

CAS: Column Address Strobe
DRAM Commands and Timing

**FIGURE 11.8:** A read cycle.

**FIGURE 11.14:** Consecutive column-read commands to different rows of the same bank: best-case scenario.
Figure 7.5: DIMMs, ranks, banks, and arrays. A system has potentially many DIMMs, each of which may contain one or more ranks. Each rank is a set of ganged DRAM devices, each of which has potentially many banks. Each bank has potentially many constituent arrays, depending on the part’s data width.
DRAM Memory-Access Protocol

1. Command transport and decode
2. In bank data movement
3. In device data movement
4. System data transport

FIGURE 11.1: Command and data movement on a generic SDRAM device.
## Timing parameters in DRAM Protocols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{AL}$</td>
<td>Added Latency to column accesses, used in DDRx SDRAM devices for posted CAS commands.</td>
</tr>
<tr>
<td>$t_{BURST}$</td>
<td>Data burst duration. The time period that data burst occupies on the data bus. Typically 4 or 8 beats of data. In DDR SDRAM, 4 beats of data occupy 2 full clock cycles.</td>
</tr>
<tr>
<td>$t_{CAS}$</td>
<td>Column Access Strobe latency. The time interval between column access command and the start of data return by the DRAM device(s). Also known as $t_{CL}$.</td>
</tr>
<tr>
<td>$t_{CCD}$</td>
<td>Column-to-Column Delay. The minimum column command timing, determined by internal burst (prefetch) length. Multiple internal bursts are used to form longer burst for column reads. $t_{CCD}$ is 2 beats (1 cycle) for DDR SDRAM, and 4 beats (2 cycles) for DDR2 SDRAM.</td>
</tr>
<tr>
<td>$t_{CMD}$</td>
<td>Command transport duration. The time period that a command occupies on the command bus as it is transported from the DRAM controller to the DRAM devices.</td>
</tr>
<tr>
<td>$t_{CWD}$</td>
<td>Column Write Delay. The time interval between issuance of the column-write command and placement of data on the data bus by the DRAM controller.</td>
</tr>
<tr>
<td>$t_{FAW}$</td>
<td>Four (row) bank Activation Window. A rolling time-frame in which a maximum of four-bank activation can be engaged. Limits peak current profile in DDR2 and DDR3 devices with more than 4 banks.</td>
</tr>
<tr>
<td>$t_{OST}$</td>
<td>ODT Switching Time. The time interval to switching ODT control from rank to rank.</td>
</tr>
<tr>
<td>$t_{RAS}$</td>
<td>Row Access Strobe. The time interval between row access command and data restoration in a DRAM array. A DRAM bank cannot be precharged until at least $t_{RAS}$ time after the previous bank activation.</td>
</tr>
<tr>
<td>$t_{RC}$</td>
<td>Row Cycle. The time interval between accesses to different rows in a bank. $t_{RC} = t_{RAS} + t_{RP}$.</td>
</tr>
<tr>
<td>$t_{RCD}$</td>
<td>Row to Column command Delay. The time interval between row access and data ready at sense amplifiers.</td>
</tr>
<tr>
<td>$t_{RFC}$</td>
<td>Refresh Cycle time. The time interval between Refresh and Activation commands.</td>
</tr>
<tr>
<td>$t_{RP}$</td>
<td>Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.</td>
</tr>
</tbody>
</table>
## DRAM Timing Parameters

<table>
<thead>
<tr>
<th></th>
<th>DDR4-2400</th>
<th>DDR3-1866</th>
<th>RLDRAM3</th>
<th>LPDRAM2</th>
<th>GDDR5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Rate</strong></td>
<td>2400 MT/s</td>
<td>1866 MT/s</td>
<td>1600 MT/s</td>
<td>800 MT/s</td>
<td>8.0 Gb/s</td>
</tr>
<tr>
<td><strong>tRC = tRP + tRAS</strong></td>
<td>45.35ns</td>
<td>47.91ns</td>
<td>12ns</td>
<td>60ns</td>
<td>-</td>
</tr>
<tr>
<td><strong>tRCD - tRP - tCL</strong></td>
<td>16-16-16</td>
<td>13-13-13</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>tRCD</strong></td>
<td>13.32ns</td>
<td>13.91ns</td>
<td>-</td>
<td>18ns</td>
<td>-</td>
</tr>
<tr>
<td><strong>tRP</strong></td>
<td>13.32ns</td>
<td>13.91ns</td>
<td>-</td>
<td>18ns</td>
<td>-</td>
</tr>
<tr>
<td><strong>tCL (tCAS)</strong></td>
<td>13.32ns</td>
<td>13.91ns</td>
<td>10ns</td>
<td>18ns</td>
<td>-</td>
</tr>
<tr>
<td><strong>tRAS</strong></td>
<td>32ns</td>
<td>34ns</td>
<td>-</td>
<td>42ns</td>
<td>-</td>
</tr>
</tbody>
</table>

### Table 1: Comparison of DRAM Technologies.

<table>
<thead>
<tr>
<th>Voltage (V)</th>
<th>SDRAM-133</th>
<th>DDR-400</th>
<th>DDR2-800</th>
<th>DDR3-800</th>
<th>DDR3-1066</th>
<th>DDR3-1333</th>
<th>DDR3-1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Capacity (per chip)</td>
<td>3.3</td>
<td>2.5</td>
<td>1.8</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Price (1GB UDIMM non-ECC)</td>
<td>$49</td>
<td>$49</td>
<td>$28</td>
<td>$75</td>
<td>$99</td>
<td>$129</td>
<td>$175</td>
</tr>
<tr>
<td>Bus Freq. (MHz)</td>
<td>133</td>
<td>200</td>
<td>400</td>
<td>400</td>
<td>533</td>
<td>667</td>
<td>800</td>
</tr>
<tr>
<td>BW (MB/s/channel)</td>
<td>1066</td>
<td>3200</td>
<td>6400</td>
<td>6400</td>
<td>533</td>
<td>10666</td>
<td>12800</td>
</tr>
<tr>
<td>t&lt;sub&gt;CK&lt;/sub&gt; (ns)</td>
<td>7.5</td>
<td>5</td>
<td>2.5</td>
<td>2.5</td>
<td>1.87</td>
<td>1.5</td>
<td>1.25</td>
</tr>
<tr>
<td>Timing (t&lt;sub&gt;CL&lt;/sub&gt;) (memory cycle)</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>Burst Length (memory cycle)</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T&lt;sub&gt;pre,T&lt;sub&gt;act,T&lt;sub&gt;col&lt;/sub&gt;, (ns)</td>
<td>22.5</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>13.5</td>
<td>13.75</td>
</tr>
<tr>
<td>T&lt;sub&gt;bi&lt;/sub&gt; (ns)</td>
<td>60</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>7.5</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
DRAM Evolution

Read Timing for Synchronous DRAM

(RAS + CAS + OE ... == Command Bus)
DRAM Evolution

Internal Structure of Virtual Channel

Bank B

Bank A
  2Kb Segment
  2Kb Segment
  2Kb Segment

Row Decoder

2Kbit

Sense Amps

Sel/Dec

16 Channels (segments)

Input/Output Buffer

# DQs

DQs

Activate

Prefetch

Read Restore

Write

Segment cache is software-managed, reduces energy
DRAM Evolution

Internal Structure of Fast Cycle RAM

SDRAM

8M Array (8Kr x 1Kb)

Row Decoder

13 bits

Sense Amps

FCRAM

8M Array (?)

Row Decoder

15 bits

Sense Amps

$t_{RCD} = 15\text{ns} \ (two \ clocks)$

$t_{RCD} = 5\text{ns} \ (one \ clock)$

Reduces access time and energy/access
Memory Controller

- Row-Buffer-Management Policy: Page Policy
  - Open-page vs. close-page

- Address Mapping Scheme
  - Minimize bank address conflicts

- Memory Transaction and DRAM Command Ordering

**Figure 13.1:** Illustration of an abstract DRAM memory controller.
Little’s Law

- Mean arrival rate: $\lambda$, Mean service rate: $\mu$, Utilization: $\rho = \lambda / \mu$
- Poisson arrivals and Exponential Service: $\lambda = \sum_{i=0}^{n} \lambda_i$
- Little’s Law: The average number of customers in the system, $L$, is equal to the average arrival rate of customer to the system, multiplied by the average system time per customer, $W$.

$$L = \lambda W$$
A Little Queue Theory

- **Mean Number in System**: \( L = \frac{\lambda}{\mu - \lambda} = \frac{\rho}{1 - \rho} \)
- **Variance of Number in System**: \( Var = \frac{\rho}{(1 - \rho)^2} \)
- **Mean Queue Length**: \( L_q = \rho L = L - \rho = \frac{\rho^2}{1 - \rho} \)
- **Average Response Time**: \( R = \frac{1}{\mu - \lambda} = \frac{1}{\mu(1 - \rho)} \)
- **Average Waiting Time**: \( W_q = \frac{\rho}{\mu - \lambda} = \frac{\rho}{\mu(1 - \rho)} \)
- **Average size of queue when it is not empty**: \( L_q' = \frac{\mu}{\mu - \lambda} = \frac{1}{1 - \rho} \)
- **Throughput, Utilization, and Traffic Intensity**: \( U = \frac{\lambda}{\mu} \)
Scheduling and Bandwidth Allocation

- Latency: Scheduling Algorithm
  - FCFS (first come, first served)
  - LCFS (last come, first served)
  - LCFS-PR (last come, first served, preempt resume)
  - SIRO (service in random order)
  - RR (round robin)
  - PS (processor sharing), Fair Queuing
  - IS (infinite server)
  - PRIO (priority scheduling for multi-customer)

- Bandwidth: Weighted Bandwidth Allocation

\[
BW(i) = \rho' \mu \cdot \left( \frac{W_i}{W_1 + W_2 + \cdots + W_n} \right)
\]
Memory Access Scheduling
- Reordering memory references to exploit locality
  - FR-FCFS Scheduling: 40% improvement
    - Load-over-store scheme: load-first
    - Access Selection: column-first (ready-first)
    - Ordered priority scheme: oldest-first
    - Precharging: close-paging

(A) Without access scheduling (56 DRAM Cycles)
Figure 4. Memory access scheduler architecture.
Table 1. Scheduling policies for the precharge managers, row arbiters, and column arbiter.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Arbiters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order</td>
<td>precharge, row, and column</td>
<td>A DRAM operation will only be performed if it is required by the oldest pending reference. While used by almost all memory controllers today, this policy yields poor performance compared to policies that look ahead in the reference stream to better utilize DRAM resources.</td>
</tr>
<tr>
<td>priority</td>
<td>precharge, row, and column</td>
<td>The operation(s) required by the highest priority ready reference(s) are performed. Three possible priority schemes include: ordered, older references are given higher priority; age-threshold, references older than some threshold age gain increased priority; and load-over-store, load references are given higher priority. Age-threshold prevents starvation while allowing greater reordering flexibility than ordered. Load-over-store decreases load latency to minimize processor stalling on stream loads.</td>
</tr>
<tr>
<td>open</td>
<td>precharge</td>
<td>A bank is only precharged if there are pending references to other rows in the bank and there are no pending references to the active row. The open policy should be employed if there is significant row locality, making it likely that future references will target the same row as previous references did.</td>
</tr>
<tr>
<td>closed</td>
<td>precharge</td>
<td>A bank is precharged as soon as there are no more pending references to the active row. The closed policy should be employed if it is unlikely that future references will target the same row as the previous set of references.</td>
</tr>
<tr>
<td>most pending</td>
<td>row and column</td>
<td>The row or column access to the row with the most pending references is selected. This allows rows to be activated that will have the highest ratio of column to row accesses, while waiting for other rows to accumulate more pending references. By selecting the column access to the most demanded row, that bank will be freed up as soon as possible to allow other references to make progress. This policy can be augmented by one of the priority schemes described above to prevent starvation.</td>
</tr>
<tr>
<td>fewest pending</td>
<td>column</td>
<td>The fewest pending policy selects the column access to the row targeted by the fewest pending references. This minimizes the time that rows with little demand remain active, allowing references to other rows in that bank to make progress sooner. A weighted combination of the fewest pending and most pending policies could also be used to select a column access. This policy can also be augmented by one of the priority schemes described above to prevent starvation.</td>
</tr>
</tbody>
</table>
Application-aware memory scheduling

Adaptive history-based arbiter
- Amount of deviation: balance of Reads and Writes
- Expected latency: minimize latency
- Adaptivity by using multiple history-based arbiters

Figure 1. The Power5 memory controller.

QoS-Aware Memory Systems

- **Smart resources**: Design each shared resource to have a configurable interference control and reduction mechanism
  - QoS-aware memory controllers
  - QoS-aware interconnects
  - QoS-aware caches

- **Dumb resources**: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system
  - QoS-aware data mapping to memory
  - QoS-aware thread scheduling to cores
Memory Scheduling for SMT. (Zhu, HPCA 2005)

FR-RCFS

- Good for single thread application
- FCFS gives unfair priority to threads that have frequent, long bursts of cache misses
- FR scheduling suffers long priority inversion blocking times due to priority chaining

![Figure 1: Memory latency and IPC for benchmark vpr when it is co-scheduled with crafty and with art](image)
The FQ memory scheduler: each thread running in a private virtual time memory system (VTMS)

- VTMS timing characteristics are time scaled in proportion to the thread’s allocated share
  - e.g. doubling tCL if the thread was allocated the half of memory system
  - virtual clock
- Scheduling algorithms
  - Earliest virtual start-time first (Fairness)
  - Earliest virtual finish-time first (QoS)

\[
S_i^k = \max \{a_i^k, F_i^{k-1}\} \quad B_j.S_i^k = \max \{a_i^k, B_j.F_i^{(k-1)}\}
\]
\[
F_i^k = S_i^k + L_i^k / \phi_i \quad B_j.F_i^k = B_j.S_i^k + B_j.L_i^k / \phi_i
\]
\[
C.F_i^k = \max \{ \max \{Ra_i, B_j.R_i\} + B.L_i^k / \phi_i, C.R_i\} + C.L_i^k / \phi_i
\]
NFQ: Fair Queuing Scheduling,

- **NFQ memory scheduler**
  - **QoS objective:** a thread allocated with a fraction of the bandwidth will run no slower than the same thread on a private memory system running at the same fraction of frequency of the shared physical memory system.
  - Earliest virtual finish-time first (VFTF)
  - **Fairness policy:** distributes excess memory system bandwidth to the thread that has consumed the least excess memory system bandwidth in the past.
  - **Priority policy**
    - Prioritize read commands
    - Prioritize CAS commands, row-buffer hit first
    - Prioritize commands with the earliest virtual finish-time
  - Preventing priority inversion
NFQ: Fair Queuing Scheduling,

Figure 3: FQ memory scheduler
The Problem: NFQ is unfairness (it’s for stateless systems without parallelism, such as network wire)

Stall-Time Fair Memory (STFM)

- To equalize the DRAM-related slowdown experienced by each thread due to interference and not unfairly penalize threads without interfering with other threads.
- Memory-slowdown = $ST_{shared} / ST_{alone}$ (ST, Stall Time)
- Determine Unfairness: The ratio between the max. slowdown and the min. Unfairness = MAX Slowdown / MIN Slowdown
- Apply Fairness-Rule:
  - If slowdown value exceeds a threshold, most slowdown first.
  - Otherwise, using FR-FCFS scheduling scheme.
    - Row-hit first
    - Oldest-first
Upsides:
- Identifies fairness as an issue in multi-core memory scheduling
- Good at providing fairness
- Being fair improves performance

Downsides:
- Does not handle all types of interference
- Somewhat complex to implement
- Slowdown estimations can be incorrect
Parallelism-Aware Batch Scheduling (PAR-BS)
- Memory-Level Parallelism (MLP)
  - Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
  - OoO, non-blocking cache, prefetching
- Effective only if the DRAM controller actually services the multiple requests in parallel in DRAM banks
- Multiple threads share the DRAM controller
- DRAM controllers are not aware of a thread’s MLP
  - Can service each thread’s outstanding requests serially, no in parallel
Parallelism-Aware Batch Scheduling

- **Principle 1:** Schedule requests from a thread back to back
  - Preserves each thread’s bank parallelism
  - But, this can cause starvation…

- **Principle 2:** Group a fixed number of oldest requests from each thread into a “batch”
  - Service the batch before all other requests
  - Form a new batch when the current batch is done
  - Eliminates starvation, provides fairness
Principle 1: Parallelism-awareness
- Schedule requests from a thread (to different banks) back to back
- Preserves each thread’s bank parallelism
- But, this can cause starvation...

Principle 2: Request Batching
- Group a fixed number of oldest requests from each thread into a “batch”
- Service the batch before all other requests
- Form a new batch when the current one is done
- Eliminates starvation, provides fairness
- Allows parallelism-awareness within a batch
## Example Within-Batch Scheduling Order

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>T3</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T0</td>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T1</td>
<td>T0</td>
<td>T3</td>
</tr>
<tr>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

### Baseline Scheduling Order (Arrival order)

<table>
<thead>
<tr>
<th>Time</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PAR-BS Scheduling Order

<table>
<thead>
<tr>
<th>Time</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Stall times

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### AVG: 5 bank access latencies

### Ranking: T0 > T1 > T2 > T3

### AVG: 3.5 bank access latencies
PAR-BS Scheduling Policy

- (1) Marked requests first
- (2) Row-hit requests first
- (3) Higher-rank thread first (shortest stall-time first)
- (4) Oldest first

Three properties:
- Exploits row-buffer locality **and** intra-thread bank parallelism
- Work-conserving
  - Services unmarked requests to banks without marked requests
- Marking-Cap is important
  - Too small cap: destroys row-buffer locality
  - Too large cap: penalizes memory non-intensive threads
PAR-BS Pros and Cons

**Upsides:**

- Identifies the problem of bank parallelism destruction across multiple threads
- Simple mechanism
- Fairness 1.11X and system throughput 8.3%

**Downsides:**

- Does not always prioritize the latency-sensitive applications → lower overall throughput
- Implementation in multiple controllers needs coordination for best performance → too frequent coordination since batching is done frequently
Problem: DRAM controllers difficult to design: It is difficult for human designers to design a policy that can adapt itself very well to different workloads and different system conditions.

Idea: Design a memory controller that adapts its scheduling policy decisions to workload behavior and system conditions using machine learning.

Observation: Reinforcement learning maps nicely to memory control.

Design: Memory controller is a reinforcement learning agent that dynamically and continuously learns and employs the best scheduling policy.
Self-Optimizing Memory Controllers
Ipek, Mutlu, et al., ISCA 2008

Figure 2: (a) Intelligent agent based on reinforcement learning principles; (b) DRAM scheduler as an RL-agent
Self-Optimizing Memory Controllers
Ipek, Mutlu, et al., ISCA 2008

Figure 4: High-level overview of an RL-based scheduler.

Figure 7: Performance comparison of in-order, FR-FCFS, RL-based, and optimistic memory controllers.
Prefetch-aware Memory Scheduling, 
Lee, et al. MICRO 2008

Idea:

- Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system.
- Carefully controlling and prioritizing prefetch requests improves performance and fairness.

Figure 3. Prefetch-Aware DRAM Controller

Rule 1 Adaptive Prefetch Scheduling (APS)

1. Critical request (C): Demand and useful prefetches are prioritized.
2. Row-hit request (RH): Row-hits are prioritized over row-conflicts.
3. Urgent request (U): Demand requests generated by cores with low prefetch accuracy are prioritized over other requests.
4. Oldest request (FCFS): Older requests are prioritized over newer ones.
ATLAS, Kim, et al. HPCA 2010

- ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling)
- The system with multiple memory controllers
- Prioritize thread whose memory episode will end soonest
  - Minimizes time spent in memory episodes across all threads
  - Supported by queueing theory:
    - Shortest-Remaining-Processing-Time scheduling is optimal in single-server queue
**Uncoordinated Controllers**

**Coordinated Controllers**

---

**Memory Service Timeline**
- **MC0**
  - T0-Req
  - T0-Req
  - T1-Req
- **MC1**
  - T0-Req
  - T1-Req
  - T0-Req

**Thread Execution Timeline**
- **T0**
  - STALL
- **T1**
  - STALL

---

**Memory Service Timeline**
- **MC0**
  - T1-Req
  - T0-Req
  - T0-Req
- **MC1**
  - T0-Req
  - T0-Req
  - T0-Req

**Thread Execution Timeline**
- **T0**
  - STALL
- **T1**
  - STALL

---

*Figure 1. Conceptual example showing the importance of coordinating the actions of multiple memory controllers*
**Remaining Service Predictor**

**Rule 1** ATLAS: Request prioritization in each memory controller

1. **TH—Over-threshold-requests-first**: Requests that have been outstanding for more than $T$ cycles in the memory controller are prioritized over all others (to prevent starvation for a long time).
2. **LAS—Higher-LAS-rank-thread-first**: Requests from threads with higher LAS-based-rank are prioritized over requests from threads with lower LAS-based rank (to maximize system performance and preserve per-thread bank-level parallelism).
3. **RH—Row-hit-first**: Row-hit requests are prioritized over row-conflict/closed requests (to exploit row buffer locality).
4. **FCFS—Oldest-first**: Older requests are prioritized over younger requests.

**Rule 2** ATLAS: Coordination at the end of a quantum

1: Each MC sends each thread’s local attained service (AS) in the last quantum to a central meta-controller. Afterwards, AS value is reset.
2: Meta-controller accumulates all local AS values for each thread and updates the TotalAS value of each thread according to Equation 1.
3: Meta-controller ranks threads based on TotalAS values; threads with lower TotalAS values are ranked higher.
4: Meta-controller broadcasts the ranking to all controllers.
5: Each MC updates thread ranks when it receives the broadcast ranking. New ranking is used for scheduling in the next quantum.
ATLAS Pros and Cons

- **Upsides:**
  - Good at improving overall throughput (compute-intensive threads are prioritized)
  - Low complexity
  - Coordination among controllers happens infrequently
  - System throughput: 8.4%

- **Downsides:**
  - Lowest/medium ranked threads get delayed significantly → high unfairness
Goal: providing the best performance and fairness at the same time

Figure 1. Performance and fairness of state-of-the-art scheduling algorithms. Lower right corner is the ideal operation point.
Throughput vs. Fairness

- **Throughput biased approach**
  - Prioritize less memory-intensive threads
  - **Memory-intensive threads**: Starvation $\rightarrow$ Unfairness

- **Fairness biased approach**
  - Take turns accessing memory: fair queueing
  - Not prioritize less memory-intensive threads: reduced throughput
  - Less memory-intensive threads always means latency-intensive threads

- **Shuffle thread ranking**
Thread Cluster Memory Scheduling

1. Group threads into two clusters (total order based on MPKI)
2. Prioritize non-intensive cluster
3. Different policies for each cluster
TCM: Scheduling Algorithm

1. **Highest-rank**: Requests from higher ranked threads prioritized
   - Non-Intensive cluster > Intensive cluster
   - Non-Intensive cluster: lower intensity ➔ higher rank
   - Intensive cluster: rank shuffling

2. **Row-hit**: Row-buffer hit requests are prioritized

3. **Oldest**: Older requests are prioritized
TCM Pros and Cons

Upsides:
- Provides both high fairness and high performance

Downsides:
- Scalability to large buffer sizes?
- Effectiveness in a heterogeneous system?
Threads in a multithreaded application are inter-dependent

Some threads can be on the critical path of execution due to synchronization; some threads are not
  - Locks and barriers

How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them

Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a parallel for loop
Heterogeneous CPU-GPU systems
- Require memory schedulers with large request buffer

Stage Memory Scheduling (SMS)
- Batch formation: maintains row buffer locality
- Batch scheduler: reduces interference between applications
- DRAM command scheduler: issues requests to DRAM

Compared to state-of-the-art memory schedulers
- Significantly simpler and more scalable (FIFO vs. CAM)
- Higher performance and fairness
CPU/GPU Heterogeneous Multi-Core

Batch Scheduler
  - Short-Job-first (SJF)
  - Round-robin

Complexity: 66% less area and 46% less power

Figure 4. Organization of the SMS.
c(O)operative thread array a(W)are warp schedu(L)ing policy
- Bank-Level Parallelism

4 policies
- CTA-aware two-level warp scheduling (MICRO 2011)
- CTA-aware Locality aware warp scheduling (CCWS)
- BLP aware warp scheduling (PAR-BS)
- Opportunistic memory-side prefetching (ISCA-2013)

Figure 5. An example illustrating (A) the under-utilization of DRAM banks with CTA-Aware-Locality, (B) improved bank-level parallelism with CTA-Aware-Locality-BLP, (C1, C2) the positive effects of Opportunistic Prefetching.

Load Criticality Aware Memory Scheduler
Ghose, et al., ISCA 2013

- Processor-side load criticality predictor
- MaxStallTime scheme

Figure 2: Overview of Commit Block Predictor (CBP) operation. Solid gray lines represent the per-cycle register counter and updates, dashed red lines illustrate a write to the CBP table (when a stalled load commits), green dash-dot-dot lines show the table lookup for loads to be issued, and blue dash-dot lines depict load issue to memory.
Interference/Slowdown Estimation

- **MISE, HPCA 2013**
  - Memory-interference Induced Slowdown Estimation (MISE)
  - Observation: An application’s performance is correlated with its memory request service rate.
  - MISE periodically gives the application’s requests the highest priority in accessing main memory

- **ASM, MICRO 2015**
  - Application Slowdown Model (ASM)
  - Accurately estimates application slowdowns due to interference at both the shared cache and main memory
  - Observation: the performance of each application is roughly proportional to the rate at which it accesses the shared cache.
  - Using an auxiliary tag store to determine the number of shared cache misses that would have been hits if the application did not share the cache with other applications

Execution-based Memory-Side Prefetching
Hashemi, et al., and Y. Patt, ISCA-43’16

- Processing-in-Memory (PIM) vs. Near-Memory Computing
- Dependent Cache Miss: Pointer Chasing
- Enhanced Memory Controller
  - The number of operations between a cache miss and its dependent cache miss is usually small.
  - Using a dynamic dataflow walk to identify these operations.
  - The dependent chain is executed in EMC to prefetch data

Figure 8: The microarchitecture of the EMC.

Summary

Memory Access Scheduling
- Row-buffer locality
- Bank-level parallelism
- QoS-aware scheduling: interference in multicore

Innovations
- FR-FCFS: row-hit first; oldest-first
- NFQ: fair queuing
- STFM: interference, most stall-time first
- PAR-BS: memory-level parallelism, batch
- ATLAS: multiple memory controller
- TCM: latency-sensitive application, thread cluster
- SMS: CPU-GPU, low-complexity FIFO